Zadanie 8

Treść

Operacja `swap(i, j)` na permutacji powoduje przestawienie elementów znajdujących się na pozycjach i oraz j. Koszt takiej operacji określamy na $|i - j|$. Kosztem ciągu operacji `swap` jest suma kosztów poszczególnych operacji.

Ułóż algorytm, który dla danych π oraz σ - permutacji liczb $\{1, 2, ..., n\}$, znajdzie ciąg operacji `swap` o najmniejszym koszcie, który przekształca permutację π w permutację σ.

Redukcja problemu

Problem transformacji permutacji π w σ można sprowadzić do problemu posortowania pewnej permutacji pomocniczej p do permutacji identycznościowej $(1, 2, ..., n)$.

Zdefiniujmy permutację p w następujący sposób: p_i jest docelową pozycją elementu, który w permutacji π znajduje się na pozycji i. Elementem na pozycji i w π jest π_i. Jego docelową pozycją w permutacji σ jest takie j, że $\sigma_j = \pi_i$. Innymi słowy, $j = \sigma^{-1}(\pi_i)$.

Zatem permutacja, którą będziemy sortować, to:

$$p_i = \sigma^{-1}(\pi_i) \text{ dla } i = 1, 2, ..., n$$

Naszym celem jest teraz znalezienie sekwencji operacji `swap` na indeksach tablicy reprezentującej p, która przekształci ją w permutację identycznościową ($p_i = i$ dla wszystkich i) i zminimalizuje sumaryczny koszt $\sum|i - j|$.

Algorytm

1. Oblicz tablicę `pos_sigma`, gdzie `pos_sigma[v]` przechowuje pozycję wartości v w permutacji σ.
2. Stwórz permutację p do posortowania: dla $i = 1, ..., n$, ustaw
$$p_i = \texttt{pos\_sigma}[\pi_i]$$
3. Dla i od 1 do n:
   i. Jeśli $p_i < i$:
      a. Zainicjalizuj pusty stos `sciezka`
      b. Ustaw `cel :=` p_i
      c. Dopóki `cel` $< i$:
      ```
      sciezka.push(cel)
      cel = p[cel]
      ```
      d. Ustaw `poprzedni` $= i$.
      e. Dopóki stos `sciezka` nie jest pusty:
      - `aktualny = sciezka.pop()`
      - Wykonaj operację `swap(poprzedni, aktualny)` na permutacji p.
      - `poprzedni = aktualny`

Opracował: Filip Figzał, 2025

Dowód poprawności

Dowód poprawności przeprowadzimy, korzystając z niezmiennika pętli.

> Po zakończeniu i-tej iteracji pętli głównej (dla ustalonego i), dla każdego indeksu $j \leq i$ zachodzi warunek $p_j \geq j$ (żadny element na lewo od i, nie chce „iść" w lewo).

Inicjalizacja

Przed pierwszą iteracją $i = 0$, niezmiennik jest trywialnie prawdziwy.

Utrzymanie

1. $\boldsymbol{p_i \geq i}$:
   W tej sytuacji warunek z niezmiennika jest spełniony dla $j = i$, a dla $j < i$ był spełniony z założenia utrzymania. Niezmiennik pozostaje prawdziwy.

2. $\boldsymbol{p_i < i}$:
   Niech p oznacza stan permutacji przed operacjami, a p' stan po operacjach. Algorytm znajduje ścieżkę $c_1, c_2, ..., c_k$ taką, że $c_1 = p_i$, $c_2 = p_{c_1}$, itd. Sekwencja operacji `swap` wykonuje cykliczne przesunięcie elementów na pozycjach $(i, c_1, ..., c_k)$. Element z pozycji i trafia na c_1, z c_1 na c_2, itd., aż element c_k trafia na i.

   W efekcie, dla każdego $m \in \{1, ..., k\}$, n a pozycję c_m trafia element, którego celem (w permutacji p) była właśnie pozycja c_m. Formalnie:

   - Na pozycję c_1 trafia element z pozycji i. Jego celem było $p_i = c_1$. Zatem nowa wartość w permutacji $p'_{c_1} = c_1$.
   - ...i tak dalej, aż do $p'_{c_k} = c_k$.

   Po tej operacji, dla każdego $m \in \{1, ..., k\}$, nowa wartość $p'_{c_m} = c_m$. Ponieważ $c_m < i$ warunek $p'_{c_m} \geq c_m$ jest spełniony. Co więcej, te pozycje nie będą już nigdy modyfikowane. Nowa wartość na pozycji i to $p'_i = p_{c_k}$, a z definicji ścieżki wiemy, że $p_{c_k} \geq i$. Zatem niezmiennik jest zachowany.

Terminacja

Po zakończeniu pętli dla $i = n$, wiemy, że dla wszystkich $j \in \{1, ..., n\}$ zachodzi $p_j \geq j$. Ponieważ p jest permutacją zbioru $\{1, ..., n\}$, jedyną możliwością jest $p_j = j$ dla wszystkich j. Permutacja jest posortowana.

Opracował: Filip Figzał, 2025

Dowód minimalności kosztu

Całkowity koszt sortowania jest sumą kosztów wszystkich wykonanych operacji `swap`:

$$\text{koszt całkowity} = \sum_{\text{operacje}} |i - j|$$

Koszt ten można interpretować jako sumaryczną odległość, o jaką przesuwane są wszystkie elementy. Minimalny koszt jest osiągany wtedy, gdy każdy element przemieszcza się od swojej pozycji początkowej do docelowej po najkrótszej możliwej drodze, czyli monotonicznie (zawsze w lewo lub zawsze w prawo, nigdy w obu kierunkach).

Obserwacja 1

Sumaryczny koszt przemieszczenia elementu z pozycji s na pozycję t za pomocą ciągu zamian jest nie mniejszy niż $|s - t|$. Równość zachodzi wtedy, gdy każda zamiana z udziałem tego elementu przesuwa go w stronę jego celu.

Obserwacja 2

Nasz algorytm zapewnia monotoniczność ruchów:

1. Gdy element na pozycji k musi przesunąć się w **lewo** (tj. $p_k < k$), algorytm uruchamia się, gdy pętla główna osiągnie $i = k$. Element ten jest przesuwany w lewo w ramach cyklicznej zamiany i ląduje na swojej docelowej pozycji p_k. Ponieważ pętla główna już minęła indeks p_k, element ten nie zostanie nigdy więcej poruszony.
2. Gdy element na pozycji k musi przesunąć się w **prawo** (tj. $p_k > k$), może on zostać przesunięty, zanim pętla główna dojdzie do $i = k$. Stanie się tak, jeśli inny element będzie musiał zająć pozycję k. Taka zamiana `swap(j, k)` będzie zainicjowana z pozycji $j > k$. Element z pozycji k zostanie przesunięty na pozycję j, czyli dalej w prawo, co jest zgodne z jego docelowym kierunkiem ruchu.

Ponieważ każdy element porusza się wyłącznie w kierunku swojej pozycji docelowej, całkowity koszt jest sumą minimalnych odległości, jakie każdy element musi pokonać, co jest zgodne z jego docelowym kierunkiem ruchu. Algorytm realizuje zatem sortowanie o minimalnym koszcie.

Opracował: Filip Figzał, 2025

Analiza złożoności obliczeniowej

1. Obliczenie permutacji p (w tym tablicy `pos_sigma`)

   $O(n)$

2. **Główna pętla**:

   Pętla zewnętrzna wykonuje się n razy.

   Rozważmy pojedyncze wejście do bloku `if` w i-tej iteracji pętli. Niech p oznacza stan permutacji przed zamianami, a p' to stan po. Pętla `while cel < i` buduje na stosie `sciezka` ciąg indeksów $c_1, ..., c_k$. Koszt znalezienia tej ścieżki i wykonania k operacji `swap` jest proporcjonalny do jej długości, czyli wynosi $O(k)$.

   Z konstrukcji pętli `while` wynikają dwie kluczowe właściwości:

   i. Dla każdego elementu ścieżki (z wyjątkiem ostatniego) zachodzi relacja: $p_{c_m} = c_{m+1}$. Zatem mamy zagwarantowane:

   $$p_i = c_1, \quad p_{c_1} = c_2, \quad ..., \quad p_{c_{k-1}} = c_k$$

   ii. Pętla kończy się, ponieważ `cel` zostaje ustawione na p_{c_k}, a ta wartość spełnia warunek $p_{c_k} \geq i$.

   Sekwencja operacji `swap` wykonuje cykliczne przesunięcie elementów, które pierwotnie znajdowały się na pozycjach $(i, c_1, ..., c_k)$. Prześledźmy, jak to wpływa na nową permutację p':

   i. Na pozycję c_1 trafia element, który pierwotnie był na pozycji i. Jego celem było $p_i = c_1$. Zatem nowa wartość w permutacji $p'_{c_1} = c_1$. Pozycja c_1 została naprawiona.

   ii. Na kolejnych pozycjach (idąc do c_k) będzie podobnie.

   iii. Na pozycję i trafia element, który pierwotnie był na pozycji c_k. Jego celem było p_{c_k}. Jeśli przypadkiem $p_{c_k} = i$, to również pozycja i zostanie naprawiona, ale nie jest to gwarantowane.

   W rezultacie, po wykonaniu k operacji `swap`, naprawiamy przynajmniej k pozycji.

   Pozycja j, która raz została naprawiona (tzn. $p_j = j$), już nigdy nie zostanie zmodyfikowana, ponieważ pętla główna nie wraca do przetworzonych indeksów, a warunek $p_j < j$ nie będzie już dla niej spełniony. Dodatkowo żaden inny element nie może prowadzić na pozycję elementu na właściwym miejscu.

   Zatem, każda z n pozycji w permutacji może zostać naprawiona (trafi na swoje miejsce) dokładnie raz, a praca wykonana wewnątrz bloków `if` jest bezpośrednio związana z naprawianiem pozycji. Skoro za koszt $O(k)$ naprawiamy przynajmniej k unikalnych pozycji, to łączny koszt naprawienia wszystkich n pozycji w trakcie całego działania algorytmu nie przekroczy $O(n)$.

   $O(n)$

Zatem, łączna złożoność wynosi: $O(n) + O(n) = O(n)$.

Złożoność pamięciowa także wynosi $O(n)$, ze względu na przechowywanie permutacji i stosu.

Opracował: Filip Figzał, 2025